Efficient Serverless Support for Multi-Instance GPUs Through Pipelining

Xinning Hui
North Carolina State University
xhui@ncsu.edu

Yuanchao Xu
University of California, Santa Cruz
yxu314@ucsc.edu

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

Abstract

Recent advancements in serverless computing have garnered interest in deploying machine learning (ML) inference tasks due to the ease of autoscaling and pay-as-you-go billing. Despite the trend of integrating GPUs in serverless platforms, existing solutions face significant GPU underutilization. This paper presents FluidFaaS, a solution that enables flexible Multi-Instance GPU (MIG) management for serverless ML. Through a novel programming system support, FluidFaaS enables fine-grained resource assignment to the components within a serverless function, based on which, it equips the invokers with runtime support that constructs pipelines on MIGs on the fly for a serverless function. Evaluations demonstrate that FluidFaaS outperforms the state-of-the-art solution, ESG, by 25%-75% in throughput while achieving up to 90% higher SLO hit rates in various workloads.

1 Introduction

Recent years have witnessed a rapidly growing interest in machine learning (ML) inferences on serverless platforms, thanks to the ease of programming and maintenance, autoscaling, and pay-as-you-go billing of serverless computing [9, 10, 12, 15, 17, 18, 25, 36–39]. There have been some recent research efforts toward efficient and low-cost GPU support for ML inference on serverless platforms. Some studies have leveraged NVIDIA Multi-Process Service (MPS) [3] to facilitate GPU sharing across different function instances [12, 18, 31, 38, 40]. However, context sharing in MPS is subject to lack of isolation, and hence, performance interference and security concerns. To overcome these limitations, recent studies [21, 25] and container orchestration systems (e.g., Kubernetes) [4] prefer Multi-Instance GPU (MIG) technology [5], which partitions a single GPU into multiple isolated MIG slices. Each MIG slice executes a function, allowing concurrent sharing of a GPU with strong isolation and no performance interference.

Although existing studies have developed GPU-aware solutions, including dynamic batching [8, 38], GPU-aware scheduling [21], and shared GPU contexts [37], they treat GPUs merely as another type of resource and apply CPU-centric serverless designs, leading to GPU under-utilization. For instance, ESG [21], a state-of-the-art work on serverless MIGs, demands 167% more than the required resource on average. First, ESG overlooks the unique challenges posed by the static partitioning of GPU into MIGs. If instance A occupies a MIG slice, and the remaining MIG slices on this GPU remain idle unless another instance requiring the same MIG slices. However, CPU-centric design in ESG does not explore how to utilize the remaining MIG slides effectively.

To address resource idle due to static MIG partitions, we propose a pipeline-based instance construction. Rather than treating an entire serverless function as a single unit for resource configuration, we introduce FluidFaaS function, a new form of serverless function that allows the runtime to automatically split a serverless function into stages. These stages are assigned to available fragmented MIG slices, creating a pipeline that utilizes these slices to serve requests, thereby improving overall GPU utilization.

Designing an efficient system to support this idea involves substantial challenges in serverless architectures. The serverless programming model, in conjunction with resource scheduling, lacks mechanisms for dynamically constructing functions based on available isolated resources. Existing serverless models require users to develop applications as static functions, which are then encapsulated within containers. These systems allocate resources to deploy instances from these predefined containers, disallowing the dynamic construction of pipeline functions and containers. Efficient use of the possibly multiple instances of a serverless function introduces additional scheduling challenges. Instances may share resources temporarily or utilize different resources and pipeline partitions to construct the pipeline. These instances exhibit varied latency and throughput. How to effectively schedule requests and auto-scale to optimize resource utilization while meeting SLOs presents a complex, multi-dimensional scheduling dilemma.

We design FluidFaaS, a flexible serverless platform for ML

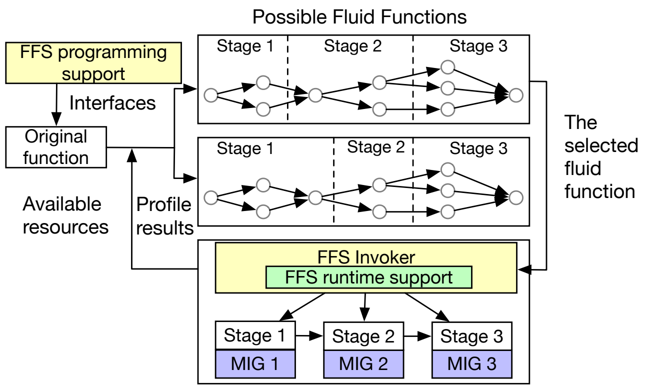


Figure 1: Overview of FluidFaaS.

inference on MIGs. FluidFaaS enables the new capability of a serverless programming model by providing programming and runtime support that enables the dynamic construction of pipelines and containers, effectively utilizing idle MIG slices.

Overall, this work makes the following major contributions:

- It identifies resource idleness as the key factor in GPU MIG underutilization.
- It designs FluidFaas, a flexible serverless platform for ML inference on MIGs by introducing pipelining-based programming support to improve GPU utilization.
- It empirically confirms the effectiveness of FluidFaas, outperforming the state-of-the-art work ESG by 25%-75% in system throughput while achieving up to 90% higher SLO hit rates in various workloads.

2 Design of FluidFaaS

This section introduces the design of FluidFaaS on how to improve the GPU resource utilization under the resource fragmentation and exclusive keep-alive state caused by MIG.

The serverless functions we target in this work are those composed of multiple Deep Neural Network (DNN) models that form a Directed Acyclic Graph (DAG). Such serverless applications are the main targets in serverless ML [13, 22, 26, 27, 31, 41, 44].

2.1 Overview

We introduce FluidFaaS (FFS) to effectively support pipeline-based instance construction. FFS comprises two primary components: programming support and the FluidFaaS runtime system.

We propose a new form of serverless function called *FluidFaaS function*. As depicted in Figure 1, the FluidFaaS programming support enables the automatic configuration of a pipelined DAG, based on available resources and profiling results. The selected FluidFaaS (FFS) function has several stages, allowing the runtime to allocate MIG slices to different stages efficiently. The FFS runtime and Invoker use one instance to manage different stages on different MIG slices and the data transfer between them. Details about the programming support are presented in Section 2.2.

2.2 Automatic Pipeline Instance Construction

When scaling up requires launching a new instance, the construction of that instance should be flexible and adaptive based on available resources. For example, depending on resource availability, the function might be divided into a two-stage or three-stage pipeline, or configured as a non-pipeline process. However, the rigid, hard-coded nature of serverless functions typically limits dynamic pipeline construction. To address this limitation, we provide programming support that ensures MIG instance usage is both transparent to users and adaptable at runtime. Transparency here refers to an interface that abstracts the complexities of MIG management away from the serverless function developer. It is essential for both ease of use and performance due to the unpredictable availability of GPU resources.

2.2.1 Programming Support

In existing serverless function platforms, a serverless function is regarded as a monolithic unit to invokers. The key to enable flexible MIG-based GPU resource mapping and utilization is to make each component in a serverless function a unit manageable at runtime on invokers. The management includes decisions on what size of the MIG slice and which specific slide is assigned to each component. The design shall also ensure the executions of the serverless function indeed execute each of its components on the MIG slice as specified by the invoker.

Our solution is *FluidFaaS function*. This new form of serverless function extends the current serverless function with a module named *FluidFaaS*. A serverless function written with FluidFaaS (FFS) APIs will be equipped with a DAG, called FFS DAG, which represents the components in the function as nodes and dataflows among them as edges. Please note that this DAG differs from task DAGs in existing serverless computing: The former captures the computation flows *within* a serverless function, while the latter is about relations *among* serverless functions. After DAG construction, each node in the FFS DAG will carry a performance profile of the size of the required memory and the running speeds of the component on each size of MIG.

When an FFS function is being launched by an invoker, based on the FFS DAG, the invoker will first find out the appropriate assignment of MIG slices to the components in the DAG, writes the assignment into the configuration layer of the FFS function, and then launches the FFS function.

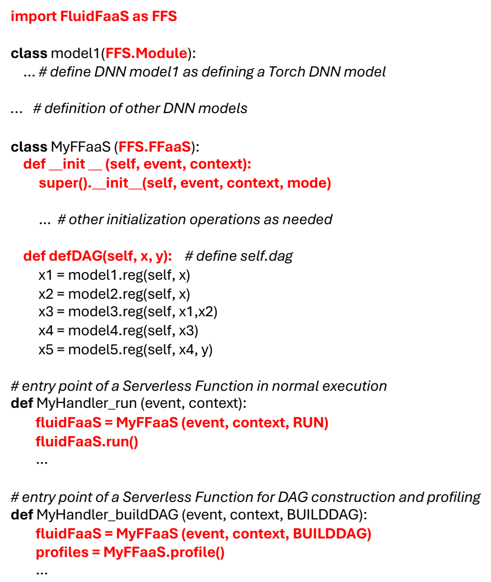
The FFS function is structured such that its execution will be done through a FluidFaaS class (a core element in the FluidFaaS module), which ensures that the execution will put each component onto the MIG slice as specified in the configuration layer, construct efficient communication channels among them, and execute the FluidFaaS function efficiently.

Programming Interface. Figure 2 gives an example showing how the programming interface of FluidFaaS can be used in writing a serverless function. FFaaS module contains two most important classes. (i) The first one is *FluidFaaS.Module*, which is a thin wrapper of the DNN model class, *nn.Module*, in PyTorch[1]. The definitions of DNN models in a PyTorch program do not need to be modified except that the superclass is changed from *nn.Module* to *FluidFaaS.Module*. Each of the five DNN models in the example is made into a subclass of *FluidFaaS.Module*. One of the main extensions of *FluidFaaS.Module* over *nn.Module* is that it includes a method "reg", which registers the model in the FFS DAG along with the types and shapes of the model's inputs and outputs (see the DAG definition function "defDAG"). (ii) The second important component in FluidFaaS module is the *FFaaS* class, which offers most of the functions that are essential for the serverless function to execute each of its components based on the MIG mapping specified by the invoker, such as build DAG, import the MIG assignments and the structure of the pipeline determined by the invoker. *FFaaS* also includes functions to profile each component. The initialization of a *FFaaS* object has two modes. In the "BUILDDAG" mode, it calls "defDAG" to build the DAG, while in the "RUN" mode, from the configuration layer, it imports the DAG, along with the MIG configs and stage configs that the invoker has put onto the configuration layer.

Runtime Support of FFaaS. Listing 1 shows the implementation of FluidFaaS.run() to explain how a FluidFaaS function gets executed. As this is "RUN" mode, the __init__() function already retrieves the DAG and the MIG assignments and pipeline configurations. The *run()* function creates a separate process for each MIG, establishes shared memory for data transfer, and sets up queues to trigger the execution of subsequent stages. The module *_run_inference()* represents the execution process for each stage. In this function, the input tensor is retrieved from shared memory, model inference is performed, and the resulting output tensor is written back to shared memory for use in the next stage. This interface is designed for simplicity, requiring only the DAG registration. The function *_terminate_processes()* monitors changes in instance states and adjusts the states of MIGs (Section 2.2.2). It also responds to termination signals from the serverless platform by stopping the processes on the MIG slices.

Unlike the default serverless function, where each function is hosted by a process that starts execution upon receiving a trigger event, FluidFaaS conducts the execution

[1]We use PyTorch as a representation; FluidFaaS is extensible to other ML frameworks.



```
import FluidFaaS as FFS

class model1(FFS.Module):
   ... # define DNN model1 as defining a Torch DNN model

...  # definition of other DNN models

class MyFFaaS (FFS.FFaaS):
   def __init__ (self, event, context):
      super().__init__(self, event, context, mode)

      ... # other initialization operations as needed

   def defDAG(self, x, y):   # define self.dag
      x1 = model1.reg(self, x)
      x2 = model2.reg(self, x)
      x3 = model3.reg(self, x1,x2)
      x4 = model4.reg(self, x3)
      x5 = model5.reg(self, x4, y)

# entry point of a Serverless Function in normal execution
def MyHandler_run (event, context):
     fluidFaaS = MyFFaaS (event, context, RUN)
     fluidFaaS.run()
     ...

# entry point of a Serverless Function for DAG construction and profiling
def MyHandler_buildDAG (event, context, BUILDDAG):
     fluidFaaS = MyFFaaS (event, context, BUILDDAG)
     profiles = MyFFaaS.profile()
     ...
```

Figure 2: Illustration of the programming of a FFaaS function.

running on a dedicated MIG instance with a separate process. These pipeline components communicate through the shared memory of the host system. The predecessor process writes its output tensor to the shared memory, as shown in the module *_write_data_to_shared_memory()*, and the successor process reads this output tensor using module *_get_data_from_shared_memory()* as its input tensor, facilitating efficient data flow between pipeline stages. This approach streamlines execution.

The use of shared memory on the host for communications helps keep the communication overhead low. We further discuss the data communication overhead associated with this design in Section 4.3, where we analyze its impact on overall system performance and efficiency.

```
class FFaaS:
    def __init__(self, event, context, mode):
      if (mode == 'RUN'):
          self.dag = self.importDAG()
          self.migs = self.importMIGs()
          self.stages = self.importStages()
          ...
      elif (mode == 'BUILDDAG'):
          ...

    def _load_models(self, DAG):
      ...
    def _run_inference(self, stage, queue, next_queue,
     shared_data, next_shared_data):
        device = torch.device("cuda")
        while True:
            input = self._get_data_from_shared_memory(
     shared_data).to(device)
            if input not empty:
                # Run all components in stage based on
     the DAG
                output = stage.run(input)
                self._write_data_to_shared_memory(
     next_shared_data, output)
                eviction = False  # Placeholder for
     actual eviction condition
```

```
                if eviction:
                    model.cpu()
                    del model
                next_queue.put()
    def _get_data_from_shared_memory(self, shared_data):
        ...
    def _write_data_to_shared_memory(self, shared_data,
     data):
        ...
    def _initialize_shared_memory_and_queues(self):
        ...
    def _start_processes(self):
        #each mig in the self.migs
            os.environ["CUDA_VISIBLE_DEVICES"] = mig
            torch.cuda.init()
            p = mp.Process(
                target=self._run_inference,
                args=(self.stages[i], self.queues[i],
     self.queues[i+1], self.shared_data[i], self.
     shared_data[i+1])
            )
            self.processes.append(p)
            p.start()
    def _terminate_processes(self):
        ...
    def profile(self):
        ...
    def run(self):
        self._initialize_shared_memory_and_queues()
        self._load_models()
        self._start_processes()
        self._terminate_processes()
        # Cleanup shared memory
        ...
```

Listing 1: Implementation of the core runtime support FFaaS

2.2.2 Runtime Support in Invoker

The programming interface introduced by the FluidFaaS module makes each component in a serverless function a unit in the FFS DAG. Based on it, the runtime support in the invoker figures out the appropriate MIG resource assignment to each of the components and outlines the appropriate pipeline for the serverless function.

Leaving these functionalities to the runtime of invokers is an important design decision of FluidFaaS. It is essential for FluidFaaS to work efficiently despite the dynamic nature of serverless environments. Workloads and system conditions can change dramatically and unpredictably; predefined pipelines for a specific application cannot suit the needs. An unbalanced pipeline can lead to inefficient resource utilization, causing bottlenecks and increased latency as certain stages may become overburdened while others remain underutilized. This solution takes into account both pipeline balance and current resource availability to ensure high performance.

This runtime support is implemented on each invoker, where it functions as a local scheduler. As the workflow depicted in Figure 1 (a), the invoker is responsible for constructing the pipeline and allocating MIG slices based on current resource availability and application knowledge which is the profile information. This decentralized approach allows the scheduler to efficiently build pipelines and allocate resources, adapting to the invoker's current conditions to ensure both efficiency and responsiveness to fluctuating workloads. Notably, this runtime system can be integrated into existing serverless architectures without requiring any modifications to the central controller, making it practical and easy to implement.

A key to building a pipeline is balance, which is crucial for performance. To accomplish this, the DAG is divided into several self-contained and balanced groups. Several studies have explored methods for partitioning DAG [21,26]. Our method is based on the dominator-based method from ESG [21] but extends it with selection with a coefficient of variation (CV). It evaluates all possible partitions of the DAG by calculating the coefficient of variation (CV) [7] to measure pipeline balance. For simplicity, consider a sequential DAG with five models: [1, 2, 3, 4, 5]. There are 2^4 possible consecutive partitions, each representing a different pipeline configuration. For instance, one configuration could be [[1], [2, 3], [4, 5]], which forms three distinct stages. The CV is determined by dividing the standard deviation of the execution times of these stages by their mean, as shown in Equation 1, where t_n denotes the execution time of stage n. If a stage involves multiple MIGs running in parallel, we use the maximum execution time among them as the stage's execution time.

$$CV = std(t_1, t_2, ..., t_n)/mean(t_1, t_2, ..., t_n) \tag{1}$$

After calculating and sorting the CVs for all possible pipeline configurations, which is done once and offline for each application (thus avoiding any runtime overhead), the next step is to assign a pipeline to the available MIGs within the invoker. We rank pipelines based on their CVs: lower CVs represent better balanced configurations. The pipelines are then evaluated in order with the profiles, against the available MIG resources. If a given pipeline can be supported by the available resources, the invoker records the pipeline configuration and corresponding MIGs on the configuration layer of the serverless function and then proceeds to launch the instance with that configuration. If the resources are insufficient, it moves to the next pipeline in the ranked list. This process continues until a suitable pipeline is found and deployed. By following this approach, we ensure that the most balanced pipeline is selected based on the available MIGs, leading to optimized performance and efficiency.

3 Evaluation Methodology

Baseline. The baseline we use for comparison is the state-of-the-art serverless ML solution supprts MIG, ESG [21]. ESG assigns functions to specific MIGs based on the resource configuration provided by the Controller. The Controller chooses the resource-efficient configuration that can meet the SLO but with the minimum resource usage.

SLO Latency Requirement. SLO latency is an important requirement on a serverless workload. It defines the acceptable latency for the platform to give a response to a request.

Table 1: Experimental hardware configuration

CPU device	4 * AMD EPYC 7763 64-Core Processors
CPU Mhz	2445.404
CPU memory	1440GB
GPU device	8 * NVIDIA A100 80GB
GPU memory	80GB

Let t be the time needed by the application to complete its entire workflow when it runs alone with a unit CPU and the minimum MIG instances, as shown in Table 3. We use *SLO scale* when describing the SLO latency of a workload. It is defined as the ratio between the SLO latency and t. By default, we set the SLO to be 3x (SLO scale=3), which refers to the case where the acceptable maximal latency is three times of t.

Table 2: Applications

Applications	**Composition**
Image classification (App 1)	Super resolution [24] ->Segmentation [2, 11] ->Classification [6, 19]
Depth recognition (App 2)	Deblur [1] ->Super resolution ->Depth recognization [30]
Background elimination (App 3)	Super resolution ->Deblur ->Background removal [29]
Expanded image classification (App 4)	Deblur ->Super resolution ->Background removal ->Segmentation ->Classification

We use the following metrics to examine the performance.
SLO Hit Rate. SLO hit rate is defined as the fraction of requests whose latencies (from the request arrives at the serverless platform and the time when the result is produced) are below the required SLO latency.
GPU time and MIG time GPU time refers to the total time the entire GPU is active, even if only one slice is being used. In contrast, MIG time specifically measures the active time of the individual MIG slice.
Evaluation environment. The runtime of FluidFaaS is implemented in the invoker node. The invoker node we use contains eight A100_PCIE_80GB GPUs. Table 1 reports the node configuration. The MIG partition for each GPU is 1g.10gb, 2g.20gb, and 4g.40gb by default. We test other partitions in the sensitive study in Section 4.4.

Table 3: Application variants and MIG slices to run

Application	**Variants**	**MIG to run (Baseline)**	**MIG to run (FluidFaaS)**
Image classification	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥1g.10gb
	large	≥3g.40gb	≥2g.20gb
Depth recognition	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥ 1g.10gb
	large	≥3g.40gb	≥2g.20gb
Background elimination	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥1g.10gb
	large	≥3g.40gb	≥2g.20gb
Expanded image classification	small	≥ 2g.20gb	≥1g.10gb
	medium	≥4g.40gb	≥1g.10gb
	large	NULL	NULL

Applications and Workloads. For head-to-head comparison, we use the four applications used in ESG [21], with each composed of multiple DNN inferences, as shown in Table 2.

Each application is available in three variants—small, medium, and large—determined by memory requirements and batch size. These variants require different sizes of MIG slices to avoid running into out-of-memory errors, as shown in Table 3. Thanks to the pipeline construction of FluidFaaS, the minimum MIG slice needed to run each variant is smaller compared to the Baseline. Notably, the 4g.40gb configuration cannot support the large variant of the *expanded image classification* application in the Baseline, so this variant is excluded from our study. We evaluate three different workloads, light, medium, and heavy, where each application is in small, medium, and large size respectively.
Traces. Following the prior work [21], we use the real-world traces from Azure Functions [34] to set the invocation frequencies and intervals of the serverless applications.

4 Evaluation Results

This section evaluates (i) the end-to-end performance of FluidFaas, compared with the state-of-the-art ESG method; (ii) its throughput and resource utilization improvement; (iii) the sensitive study for FluidFaaS with different MIG partitions.

4.1 End-to-End Performance

Table 4 shows the normalized GPU time and MIG time, SLO hit rates, P95 tail latency, and average queuing time across various workloads. The table shows that FluidFaaS improves the SLO hit rate by 90% in medium workloads and 61% in heavy workloads across all applications while achieving a similar SLO hit rate as ESG in light workloads. Additionally, FluidFaaS achieves these improvements with lower GPU time, saving 5.5% in medium workloads and 6.5% in heavy workloads. The MIG time remains comparable, with a maximum difference of 3.9%.

Figure 3 illustrates system throughput across different workloads. FluidFaaS achieves a 75% higher throughput in heavy workloads and a 25% increase in medium workloads while maintaining similar throughput in light workloads. This higher throughput allows FluidFaaS to significantly reduce queuing time, as shown in Table 4, with up to 83.3% reduction for expanded image classification in heavy workloads compared to ESG. For other applications, it achieves at least a 50% reduction. This decrease in queuing time directly affects the P95 tail latency, also shown in Table 4. Specifically, FluidFaaS improves the P95 tail latency by up to 81% in heavy workloads and 70% in medium workloads while maintaining similar latency in light workloads.

4.2 Detailed Analysis

As described in Sec.3, each GPU is divided into 4g.40gb, 2g.20gb, and 1g.10gb slices. In a light workload, all slices can host functions, resulting in similar performance between

Table 4: End-to-end Performance

Workload	Method	Norm. time		SLO hit rate (%)				P95 tail latency (ms)				Queuing Time (ms)			
		GPU	MIG	App 1	App 2	App 3	App 4	App 1	App 2	App 3	App 4	App 1	App 2	App 3	App 4
Light	ESG	1	**1**	96	96	95	96	**503**	**471**	**840**	**652**	**588**	**548**	**669**	**647**
	FluidFaaS	1	1.02	96	96	95	96	508	471	843	680	595	560	669	670
Medium	ESG	1	**1**	58	70	67	96	**971**	2693	5724	**780**	3348	1295	2123	627
	FluidFaaS	**0.946**	1.008	**94**	**96**	**95**	**96**	1067	**571**	**1713**	782	**723**	**497**	**622**	**605**
Heavy	ESG	1	**1**	52	36	52	60	11375	33941	17707	15539	4047	10471	7276	6279
	FluidFaaS	**0.935**	1.039	**70**	**43**	**54**	**94**	**7479**	**13866**	**15506**	**2983**	**2524**	**3888**	**4295**	**1049**

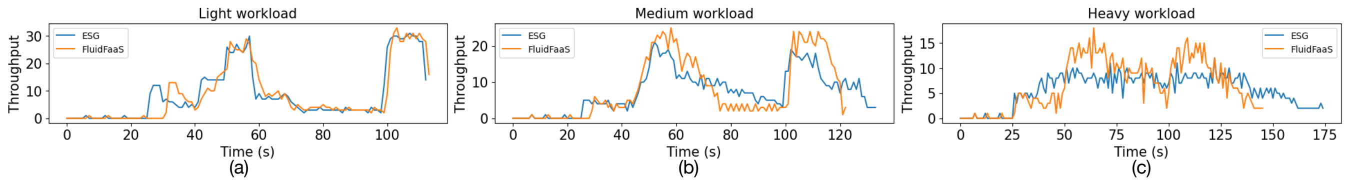


Figure 3: Throughput in different workloads.

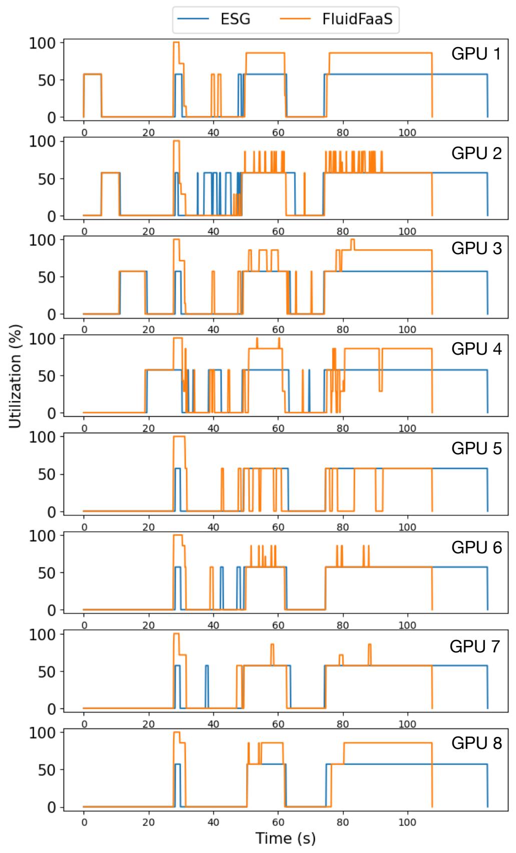


Figure 4: GPU utilization in the heavy workload.

FluidFaaS and ESG. In medium workloads, ESG cannot utilize the 1g.10gb slices due to memory limitations, whereas FluidFaaS can leverage all slices thanks to its DAG partitioning and pipeline construction. In heavy workloads, ESG can only use the 4g.40gb slices, while FluidFaaS utilizes both the 4g.40gb and 2g.20gb slices.

Figure 4 shows the GPU utilization in the heavy workload. Due to space constraints, the GPU utilization for medium and light workloads is provided in Figure 7 and Figure 8 in the Appendix. As illustrated, FluidFaaS improves GPU utilization by 75% compared to ESG during task bursts, which enables the higher throughput observed in Figure 3 (c). In this scenario, FluidFaaS can utilize a total of 7g.70gb, while ESG is limited to 4g.40gb, resulting in a 75% increase in throughput. In the medium workload, the ESG is limited to using 4g.40gb and 2g.20gb slices, while FluidFaaS can utilize all available slices, resulting in a 25% increase in throughput. The throughput is similar in the light workload scenario because each GPU has a comparable utilization level.

The high throughput allows FluidFaaS to complete all tasks 10% faster in medium workloads and 17% faster in heavy workloads compared to ESG, as shown in Figure 3(b) and (c). Thus, the earlier finish time leads to the GPU time saving. Additionally, this increased throughput results in shorter queuing times, as indicated in Table4, positively impacting both end-to-end latency and SLO hit rates.

4.3 Breakdown

Figure 5 illustrates the time breakdown of various workloads for ESG (left) and FluidFaaS (right). In the light workload, both FluidFaaS and ESG perform similarly, with execution times within the SLO limit for each application. However, in the medium and heavy workloads, FluidFaaS demonstrates up to 2.36 times lower latency than ESG. This reduction in latency for FluidFaaS leads to higher SLO hit rates, while the longer latency experienced by ESG in these workloads results in lower SLO compliance, as shown in Table 4.

It is evident that FluidFaaS consistently experiences a slightly longer data transfer overhead—ranging from 10ms to 40ms—due to communication between pipeline stages, compared to ESG's 1–5ms. Nevertheless, this overhead is offset by FluidFaaS's significantly shorter queuing times. ESG incurs much longer queuing times than FluidFaaS, with delays up to three times longer, ranging from hundreds to thousands

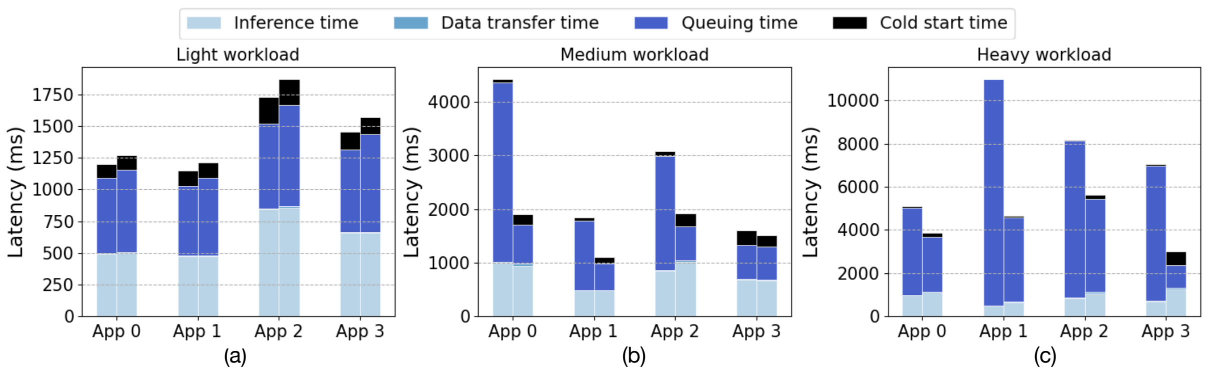


Figure 5: End-to-end latency breakdown. Left bar is for ESG and right one is for FluidFaaS.

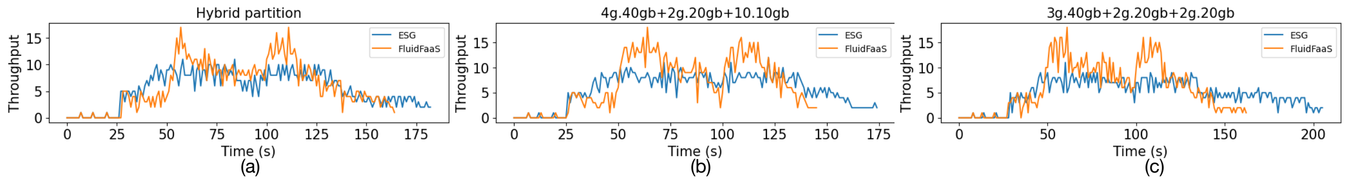


Figure 6: Throughput in different partitions.

Table 5: Different MIG partition

Name	Partition
Hybrid	1 * [1g.10gb *7] 2 * [2g.20gb*3+1g.10gb] 4 * [3g.40gb+4g.40gb] 1 * [4g.40gb+2g.20gb+1g.10gb]
P1	8 * [4g.40gb+2g.20gb+1g.10gb]
P2	8 * [3g.40gb+2g.20gb+2g.20gb]

of milliseconds, as depicted in Table 4. Therefore, the data transfer overhead introduced by the pipeline in FluidFaaS is marginal compared to the substantial benefits it brings in reducing overall latency and improving performance under heavier workloads.

4.4 Sensitive Study

In this section, we evaluate how FluidFaaS performs under different MIG partitions. In the above evaluations, each GPU is partitioned into one 4g.40gb instance, one 2g.20gb instance, and one 1g.10gb instance. We also experimented with a hybrid partitioning scheme—where each GPU has different partitions—and an alternative uniform partitioning scheme that divides each GPU into one 3g.40gb instance and two 2g.20gb instances, as detailed in Table 5.

As shown in Table 4, both FluidFaaS and ESG perform well under light workloads. However, under heavy workloads, FluidFaaS leverages fragmented resources to achieve better performance. Our sensitivity study demonstrates that FluidFaaS outperforms ESG not only in a single partition but across multiple partitions, especially when system resources are underutilized. Figure 6 illustrates that FluidFaaS surpasses ESG across all partitioning schemes. Specifically, in the hybrid partition, FluidFaaS achieves 70% higher throughput; in partitioning scheme P1, it achieves 75% higher; and in scheme P2, it reaches 78% higher throughput. These improvements stem from the fragmented small MIG slices that cannot be utilized by ESG but are effectively employed in pipelines by FluidFaaS.

5 Related Work

Several studies have explored serverless ML inference, using NVIDIA's MPS for GPU sharing to optimize batch sizes and computational resource allocation to improve throughput [18, 31, 38, 40, 42]. In these studies, all processes under MPS share the same GPU resources—including Streaming Multiprocessors (SMs), memory bandwidth, and caches—which leads to serious interferences. Also, sharing memory spaces increases the risk of unintended data access between processes, posing security concerns in multi-tenant environments. Streambox [37], which employs fine-grained CUDA Streams, faces the same performance interference and security issues.

ESG [21] proposes using NVIDIA's Multi-Instance GPU (MIG), which provides strong isolation, for GPU sharing in serverless platforms. However, ESG uses the entire serverless function as the unit for GPU resource assignment, causing serious resource fragmentation and underutilization. Miso [25] also suggested using MIG to co-run machine learning applications in cloud computing environments; however, they predefined the MIG slices and determined application colocation based on profiling data, lacking a flexible solution for dynamic co-running decisions.

Other research on GPU-based serverless computing has focused on optimizing CPU-GPU data transfers [20, 43] and reducing cold start overhead [32, 33]. Some studies [23, 28] have explored task scheduling on GPUs but treated the entire GPU as the minimal computational unit. Additionally, efforts have been made to extend serverless computing support to heterogeneous hardware, including FPGAs and NVIDIA DPUs [14, 35]. Dgsf [16] proposed disaggregated GPUs for serverless computing.

6 Conclusion

This paper proposes FluidFaaS, the first solution that enables flexible GPU MIG management of the components within a serverless ML function while minimizing resource fragmentation and underutilization. It features novel programming system support, on-the-fly pipeline construction, and GPU-aware function state management. It demonstrates significant improvements in system throughput and SLO hit rates compared to existing solutions, underscoring the importance of considering GPU-specific characteristics in serverless computing environments. It opens new opportunities for flexible resource management in GPU-based serverless platforms.

References

[1] DeblurGAN. https://github.com/pablodz/DeblurGAN.

[2] DEEPLABV3. https://pytorch.org/hub/pytorch_vision_deeplabv3_resnet101/.

[3] NVIDIA 2020, Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html.

[4] NVIDIA k8s-device-plugin. https://github.com/NVIDIA/k8s-device-plugin/.

[5] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[6] ResNet50. https://pytorch.org/hub/nvidia_deeplearningexamples_resnet50/.

[7] Hervé Abdi. Coefficient of variation. *Encyclopedia of research design*, 1(5):169–171, 2010.

[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[9] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[10] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Ra Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[11] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[12] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems*, 4:20–32, 2022.

[13] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[14] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–813, 2022.

[15] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2020.

[16] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2022.

[17] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[18] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. *arXiv preprint arXiv:2309.00558*, 2023.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Sungho Hong. *GPU-enabled Functional-as-a-Service*. PhD thesis, Arizona State University, 2022.

[21] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. Esg: Pipeline-conscious efficient scheduling of dnn workflows on serverless platforms with shareable gpus. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 42–55, 2024.

[22] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[23] Vincent Lannurien, Laurent d'Orazio, Olivier Barais, Esther Bernard, Olivier Weppe, Laurent Beaulieu, Amine Kacete, Stéphane Paquelet, and Jalil Boukhobza. Herofake: Heterogeneous resources orchestration in a serverless cloud–an application to deepfake detection. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 154–165. IEEE, 2023.

[24] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[25] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[26] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, 2022.

[27] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. Orion and the three rights: Sizing, bundling, and prewarming for serverless dags. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[28] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. Kernel-as-a-service: A serverless interface to gpus. *arXiv preprint arXiv:2212.08146*, 2022.

[29] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. U2-net: Going deeper with nested u-structure for salient object detection. volume 106, page 107404, 2020.

[30] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer, 2020.

[31] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM symposium on cloud computing*, pages 1–17, 2021.

[32] Justin San Juan and Bernard Wong. Reducing the cost of gpu cold starts in serverless deep learning inference serving. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops*

and other Affiliated Events (PerCom Workshops), pages 225–230. IEEE, 2023.

[33] Justin David Quitalig San Juan. Flashpoint: A low-latency serverless platform for deep learning inference serving. Master's thesis, University of Waterloo, 2023.

[34] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[35] Jessica Vandebon, Jose GF Coutinho, and Wayne Luk. Scheduling hardware-accelerated cloud functions. *Journal of Signal Processing Systems*, 93:1419–1431, 2021.

[36] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[37] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. Streambox: A lightweight gpu sandbox for serverless inference workflow. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 59–73, 2024.

[38] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[39] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[40] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2023.

[41] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[42] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. Astraea: towards qos-aware and resource-efficient multi-stage gpu services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 570–582, 2022.

[43] Ming Zhao, Kritshekhar Jha, and Sungho Hong. Gpu-enabled function-as-a-service for machine learning inference. *arXiv preprint arXiv:2303.05601*, 2023.

[44] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1–14, 2023.

Appendix

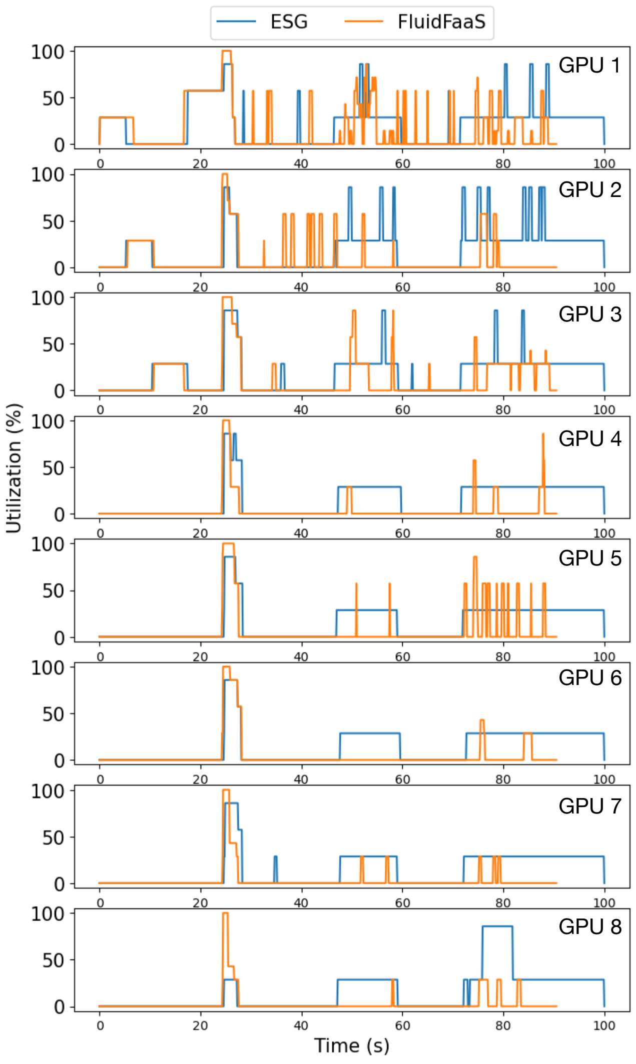


Figure 7: GPU utilization in the medium workload.

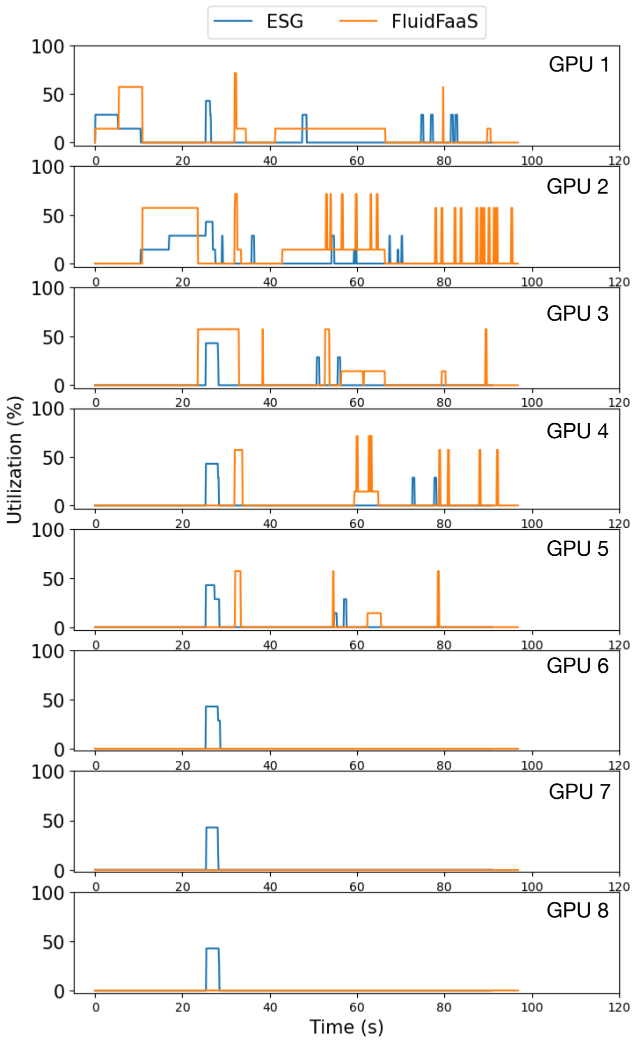


Figure 8: GPU utilization in the light workload.